# OpenMP Basics

Ned Nedialkov

McMaster University

28 February 2023

# Outline

# Syncronization

## omp critical

- the block that follows is executed by all threads
- one thread at a time

Example

```
  int x = 0, y = 0;
#pragma omp parallel num_threads(4)
  {
#pragma omp single
    x++;
#pragma omp critical
    {
      y++; x++;
    }
  }
  printf("single x=%d, critical y=%d\n", x, y);
```

outputs

```
single x=5, critical y=4
```

## `atomic`

- like critical, but protects a single read/write
- `critical` protects code, `atomic` protects memory

```
#pragma omp atomic
  x += y;
```

## `omp barrier`

- threads wait until all reach the barrier
- after the barrier, threads continue in parallel

## `omp ordered`

- ordered region executes in sequential order
- code outside it can run in parallel

## Example

```
#pragma omp parallel for ordered
  for (int i = 0; i < 8; i++)
    {
#pragma omp ordered
      {
        int thread_num = omp_get_thread_num();
        printf("iteration %i, thread %d \n", i, thread_num);
      }
    }
```

with 4 threads outputs

```
iteration 0, thread  0
iteration 1, thread  0
iteration 2, thread  1
iteration 3, thread  1
iteration 4, thread  2
iteration 5, thread  2
iteration 6, thread  3
iteration 7, thread  3
```

## nowait

- removes implicit synchronization
- usefule when there are independent regions

Examples

```
#pragma omp parallel
  {
#pragma omp sections nowait
   {
#pragma omp section
     for (i = 0; i < N; i++)
       c[i] = a[i] + b[i];
#pragma omp section
     for (i = 0; i < N; i++)
       d[i] = a[i] - b[i];
   }
  }
```

```
#pragma omp parallel
  {
#pragma omp for nowait
   for (int i = 0; i < n; i++)
     b[i] = a[i] * a[i];
#pragma omp for nowait
   for (int i = 0; i < n; i++)
     c[i] = a[i];
  }
```

# Reduction

Example: compute the sum of Fibonacci numbers in $[m, n]$

```
unsigned long fibonacci(unsigned n)
{
  if (n < 2) return n;
  unsigned long f1 = fibonacci(n - 1);
  unsigned long f2 = fibonacci(n - 2);
  return f1 + f2;
}
```

```
  unsigned long sum = 0;
#pragma omp parallel for reduction(+ : sum) schedule(dynamic)
  for (int i = m; i <= n; i++)
    sum += fibonacci(i);
```

# Taskloop

`omp taskloop`

- loop can be expressed as a collection of independent tasks, which can be executed in parallel
- number of tasks can be controlled with `grainsize`: minimum minimum number of loop iterations to be assigned to a single task
- `num_tasks` can specify the maximum number of tasks that can be created
- Consider using `taskloop` instead of `schedule(dynamic)`

Example. Compute $H(n) = 1 + 1/2 + \cdots + 1/n$.

```
double H(unsigned n)
{
  double sum = 1;
#pragma omp simd
  for (int i = 2; i <= n; i++) sum += 1.0 / n;
  return sum;
}
```

Then for given $n$, compute in parallel $H(1) + H(2) + \cdots + H(n)$.

```
  double sum = 0;
#pragma omp parallel
#pragma omp single
#pragma omp taskloop reduction(+ : sum)
  for (int i = 1; i <= n; i++) sum += H(i);
```

versus

```
  sum = 0;
#pragma omp parallel for reduction(+ : sum) schedule(dynamic)
  for (int i = 1; i <= n; i++) sum += H(i);
```

# Environment variables

`OMP_DYNAMIC = true | false`

Enable or disable the dynamic adjustment of the number of threads within a team. If `false` the system uses the specified number of threads.

`OMP_WAIT_POLICY = active | passive`

`active` waiting threads are mostly active, spin rather than sleep.
`passive` waiting threads are mostly passive.

`OMP_PROC_BIND = true | false`

`true` threads cannot move between cores.

# Tips

- Start with a well-structured serial program.
- Have good serial code.
- Profile to identify bottlenecks.
- Improve the code piece by piece, not the whole program.
- If a serial program performs poorly, how much better can it be on 8, 16, ..., cores?
- Always use `default(none)`.
- Use private data as much as possible.
- Minimize the number of parallel regions.
- Use `nowait` where feasible.
- "Play" with `schedule`.
- Avoid nested parallelism.

Consider setting

```
OMP_WAIT_POLICY=active
OMP_DYNAMIC=false
OMP_PROC_BIND=true
```